C6

```
%/*
%** Copyright 1997,1998 EMC Corporation
%*/

/*
**  Leading % causes rpcgen to pass a line directly thought to the output,
**  ie edmlink_sunrpc.h in this case.  This allows the .h to make a little
**  more sense and be properly documented.
*/

%/*
% * dispatch_daemon.x : EDM Dispatch Daemon C/S communication module
% *
% * Mission Statement:  This is an RPCGEN file which defines the RPC interface
% *                     between the Dispatch Daemon (which resides on
% *                     the EDM server) and  the backup client callers of its
% *                     functions.  This defines the RPC level calls that a
% *                     "caller" can make and the "service" will respond to.
% *
% * Primary Data Acted On: This defines the data that will flow over the wire.
% *                        The RPC mechanism will take care of data
                                                              marshalling
% *
% *
% * Compile-Time Options:
% *                     This acutally gets run through RPCGEN not compiled.
                                                                       It
% *                      must be run through with the -h flag to create a
% *                      header, the -m flag to create the service side
% *                      routines, the -l flag to create the client side
% *                      routines, and the -c flag to create the common data
% *                      marshalling routines.
% *
% * Basic idea here:
% *                     Define the RPC level interfaces to the Dispatch Daemon
% *                     and all data types that will be passed via RPC.
%*/


/*******************************************************************
 Constant Definitions
*******************************************************************/


/*******************************************************************
 Data Structure Definitions
*******************************************************************/

        struct DD_rpc_objID
        {
          int           type;    /* Object identifier (DD_OTYPE_*) */
%#define DD_OTYPE_INIT_IN        1     /* Initialize Input Object */
%#define DD_OTYPE_INIT_OUT       2     /* Initialize Output Object */
          long          len;     /* Length of structure, version number */
        };


        struct DD_client_session_id {
                unsigned long   high;
                unsigned long   low;
        };

const DD_SERVICE_RESTORE=1;
/* structures for input and output of re_initialize rpc call:  */
        struct DD_initialize_args {
                int       service;
```

```
                string hostname<>;
                string username<>;
                unsigned int timeout;
        };

const DD_SERVICE_FAILURE_NONEXEC=-4;
const DD_SERVICE_FAILURE_PERMS=-2;
const DD_SERVICE_FAILURE_EXEC=-1;
const DD_SERVICE_STARTING=1;
const DD_SERVICE_RUNNING=2;
const DD_SERVICE_COMPLETED=4;

        struct DD_initialize_result {
                unsigned int status;
                DD_client_session_id service_handle;
        };

/* structures for getstatus function */
        struct DD_getservicestatus_args {
                int                 status;
                DD_client_session_id    service_handle;
        };

        struct DD_getservicestatus_result {
                int                 status;
                opaque              handle<128>;
        };

/* work item type */
/*
 * These match the rbconfig.h for the most part. There are
 * some extras for identifying NOS workitems.
 */
const FS_BACKUP_TYPE              =       0;
const SHARED_PART_BACKUP_TYPE     =       1;
const SHARED_M_PART_BACKUP_TYPE =         2;
const OFFLINE_DB_TYPE             =       3;
const ONLINE_KICKDB_TYPE          =       4;
const ONLINE_LISTDB_TYPE          =       5;
const DCONN_KICK_TYPE             =       6;
const DCONN_NET_TYPE              =       7;
const DCONN_WRK_TYPE              =       8;

/* length of various buffers */
const MEDNAME_SIZE =    6;
const TRLNAME_SIZE =    16;
const WINAME_SIZE =     64;
const TEMPLNAME_SIZE =  64;
const USERNAME_SIZE =   64;
const HOSTNAME_SIZE =   256;
const CLNTNAME_SIZE =   64;
const SERVER_SIZE =     256;
const MAX_STRING_SIZE = 256;  /* must be the length of the longest buffer */

/* defines for operation_type */
const BACKUP_TYPE =      1;
const RESTORE_TYPE =     2;
const OTHER_TYPE =       16;

/* work item structure */
struct WIProgress {
    unsigned long       time_started;
    unsigned long       curr_time;

    unsigned long       total_kbytes_sofar;
    unsigned long       total_files;
    unsigned long       total_badfiles;

    unsigned long       curr_kbytes_sofar;
    unsigned long       curr_time_slice;
    unsigned long       curr_files;

    unsigned long       total_files_expected;
    unsigned long       total_kb_expected;

    int                 operation_type;
    int                 completed;
    unsigned long       status;

    struct WIProgress     *next;

    char                wi_name[WINAME_SIZE];
    char                trail_name[TRLNAME_SIZE];
    char                trailset_name[TRLNAME_SIZE];
    char                template_name[TEMPLNAME_SIZE];
    char                client_name[CLNTNAME_SIZE];
    char                server_name[SERVER_SIZE];
    char                media_type[MEDNAME_SIZE];
    char                userid[USERNAME_SIZE];

    char                level;
    char                type;
};

/* SUMMARY structure */
struct EDMProgress {
    unsigned long       time_started;
    unsigned long       curr_time;

    unsigned long       total_kbytes_sofar;
    unsigned long       total_files;
    unsigned long       total_badfiles;

    unsigned long       curr_time_slice;
    unsigned long       curr_kbytes_sofar;
    unsigned long       curr_files;

    unsigned long       active;
    unsigned long       total;
    unsigned long       failed;
    unsigned long       successful;

    unsigned long       total_files_expected;
    unsigned long       total_kb_expected;

    int                 operation_type;
    int                 completed;
    unsigned long       status;

    struct EDMProgress    *next;

    char                host_name[HOSTNAME_SIZE];
};

struct EDMStats
{
    unsigned long       status;
    EDMProgress         edm;
    WIProgress          *wiprogress;
};

struct CC_Notify
```

```
{
    int                 msgtype;
    int                 sourcemodule;
    int                 level;
    int                 msglen;
    string              msgtext<>;
};

struct SessionInfo
{
        DD_client_session_id    service_handle;
        unsigned int            status;
        unsigned long           jobstarttime;
        int                     operation_type;
        long                    lastSent;
        long                    lastReceived;
        int                     outhandle;
        int                     errhandle;

      SessionInfo               *next;
};

struct SessionBlock
{
        struct SessionInfo      *sess;
        int                     totalsessions;
};

program EDM_DISPATCH_DAEMON {

   version EDMDD_FUNCTIONS {

           /* Functions for EDMRST_Initialize */
           DD_initialize_result dd_initialize( DD_initialize_args ) = 1;

           DD_getservicestatus_result dd_getservicestatus(
                                       DD_getservicestatus_args ) = 2;

           SessionBlock dd_getsessioninfo( DD_getservicestatus_args ) = 3;

   } = 1; /* This is version 1 */

%/* This is the RPC program number.
                                   These are reserved in /pds/docs/RPC_numbers
% * This number cannot be re-used by any other RPC daemon on the machine,
                                                                      as it
% * identifies this daemon uniquely.  If it were to be re-used,
                                                             the last daemon
% * to register would be contacted when RPC's come in for this number.
%*/
} = 390015;
```

```
/******************************************************************************
 **
 ** File Name:   RSTinitfin.c
 **
 ** Copyright (c) 1998,1999 by EMC Corporation.
 **
 ** Purpose:
 **               This module contains the Restore API functions to
 **               initialize and terminate the restore operation.
 **
 ** Table of Contents:
 ** ------------------
 **       API Functions:
 **               EDMRST_Initialize
 **               EDMRST_Finish
 **
 **       Internal Functions:
 **
 **
 ** Compile-Time Options:
 **              This section must list any compile time definitions
 **              which will affect this header.
 **
 *****************************************************************************/


/* The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/

#ifndef lint
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define  _POSIX_SOURCE 1

/*
 * System headers.
 */
#include <pwd.h>


/*
 * Epoch headers.
 */
#include <eb/eb_port.h>
#include <eb/rb_log.h>

/*
 * Local headers
 */
#include <RSTinterns.h>
#include <RSTsup_csm.h>
```

```
/*
 * Comms headers.
 */
#include <restore/csc_EDMDispatch.h>
#include <restore/csc_EDMRestoreEng.h>
#include <restore/dispatch_daemon.h>
#include <restore/restore_engine.h>
#include <edmlink/edmlink_api.h>

/*
 * #defines, structures, typedefs local to this source file
 */


/*
 * Global declarations
 */

   internalHandlePtr handlePtr = NULL;
```

```
/*********************************************************************
 * EDMRST_Initialize:
 *
 *  This function takes care of all the initialization for a recovery
 *  session.  This must be called prior to any of the other functions
 *  in the Recover API.
 *
 * Parameters:
 *
 *  hostname   (I) - The machine name of the server to use.
 *  svrHdl     (O) - A handle to receive a pointer to this user's client
 *                   handle for the Restore Engine connection.
 *  timeout    (I) - The maximum number of seconds to wait for the connection
 *                   to the Restore Engine process to be completed.
 *
 *********************************************************************/

eerrno_ty
EDMRST_Initialize( hostname_ty   hostname,
                   serverHandle  *svrHdl,
                   unsigned long timeout )
{
        eerrno_ty api_status = E_SUCCESS;

        uid_t   human_uid;
        struct  passwd *pw;
        char    *human_uidname ;

        RE_initialize_args      re_init_args;
        RE_status_result        *re_init_result;
        rpc_if_handle_t                 re_if_spec;
        rpc_binding_handle_t    re_handle;
        int                     retval;
        time_t                  end_time;

#ifdef DEBUG
#define  RPC_TIMEOUT     3600
        struct timeval  rpc_timeout;
#endif

/********************** BEGINNING OF Dispatch Daemon STUFF ***************/
    error_status_t status;
    DD_initialize_args    initargs;
    DD_getservicestatus_args  statargs;
    DD_initialize_result  *initres = NULL;
    DD_getservicestatus_result  *statres = NULL;
    rpc_if_handle_t       if_spec;


    time( &end_time );   /* compute time to give up waiting */
    end_time += timeout;

    memset(&if_spec,0,sizeof(rpc_if_handle_t));
    memset(&re_if_spec,0,sizeof(rpc_if_handle_t));

    if (svrHdl == NULL || hostname == NULL )
    {
        return( EP_RB_RECOVER_BAD_ARGS );
    }

        rec_api_log_begin( "edmrestore_api" );
                                                /* init logs, ignore errs?? */

        /* get user name to pass to DD and RE */
        human_uid = getuid();
        pw = getpwuid( human_uid );
```

```
        if (pw == NULL || NULL == pw->pw_name )
        {
                /* Trouble.  */

                rec_api_log_csm(SUB_CSM_USER_NOT_IN_PASSWD, NULL);
                return(EP_RB_RECOVER_PERMISSION_DENIED);
        }

        human_uidname = pw->pw_name;

    handlePtr = (internalHandle *) calloc(1, sizeof(internalHandle));

    /* Use this macro to setup the interface spec */
    CLIENT_IFSPEC(if_spec);

    /* Arrive at a server binding.  Note that if they didn't give us
    ** a valid host parameter, this will fail and drop through and
    ** return NULL in the end.
    ** This call will get and store a fully resolved binding
    ** handle to the host.  The first time we ever call the host,
    ** csc_get_handle will resolve and store the binding.  If we
    ** ever use csc_get_handle to talk to the same host again,
    ** it will just give back the previously resolved binding.
    */

    retval = csc_get_handle((unsigned char *) hostname,
                            if_spec,
                            SERVER_GROUP,
                            &handlePtr -> dd_binding_handle,
                            &status);

    /*
    ** Find out if we got csc handle and see if status is bad.
    ** error_status_ok is a macro defined in cscomm.h.
    */

    if ((status != error_status_ok) || (retval == 0))
    {
      /*  If errno not set, use status if it is a valid errno value */

      if ( errno == 0 )
        errno = ( strerror( status ) ? status : ETIME );

      rec_api_log_csm( SUB_CSM_RPC_FAIL,
                       "failure finding edmdispd to start restore engine" );

      return EP_RB_RECOVER_SERVERFAIL;
    }

    errno = 0;

#ifdef DEBUG
/* increase rpc timeout during debugging */
    rpc_timeout.tv_sec = RPC_TIMEOUT;
    rpc_timeout.tv_usec = 0;
    clnt_control( handlePtr->dd_binding_handle, CLSET_TIMEOUT,
                  (char *)&rpc_timeout );
#endif

    initargs.service = DD_SERVICE_RESTORE;
    initargs.hostname = hostname;
    initargs.username = human_uidname;
    initargs.timeout = timeout;

    initres = dd_initialize_1( &initargs, handlePtr -> dd_binding_handle );
            /* Will have _1 for RPC call  */
    if (initres == NULL)
    {
        return EP_RB_RECOVER_RPC_FAIL;
    }

    statargs.service_handle = initres -> service_handle;
    statargs.status = 0;

    statres = dd_getservicestatus_1( &statargs, handlePtr->dd_binding_handle );

    if (statres == NULL)
    {
        return EP_RB_RECOVER_RPC_FAIL;
    }

    while (statres -> status == DD_SERVICE_STARTING )
    {
        time_t  now;

        xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
        time( &now );
        if (now >= end_time)
        {
                rec_api_log_csm( SUB_CSM_RPC_FAIL,
                        "timeout waiting for edmdispd to start restore engine"
                                                                            );
                return EP_RB_RECOVER_SERVERFAIL;
        }

        sleep(1);

        statres = dd_getservicestatus_1( &statargs,
                                        handlePtr -> dd_binding_handle );

        if (statres == NULL)
        {
            rec_api_log_csm( SUB_CSM_RPC_FAIL,
        "failure getting status from edmdispd while starting restore engine" );
            return EP_RB_RECOVER_RPC_FAIL;
        }
    }

    if (statres -> status != DD_SERVICE_RUNNING)
    {
        rec_api_log_csm( SUB_CSM_RPC_FAIL,
                        "edmdispd failure while starting restore engine" );
        xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
       return EP_RB_RECOVER_SERVERFAIL;
    }

    memcpy( handlePtr -> opaque128,
            statres -> handle.handle_val,
            sizeof(handlePtr -> opaque128) );

    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );

/********************** END OF Dispatch Daemon STUFF ***************/

/* Restore Engine FUNCTIONALITY BEGINS HERE */

/*       RE_CLIENT_IFSPEC(re_if_spec);  */

        retval = csc_private_ifspec_init(
                            (unsigned char *) handlePtr -> opaque128,
                           RE_PROGNUM,
```

```
                                 RE_VERSNUM,
                                  &re_if_spec,
                                  &status);

        if (retval == 0)
        {
            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                        "failure initializing interface to restore engine"
                                                                            );
            return EP_RB_RECOVER_SERVERFAIL;
        }

         api_status = EP_RB_RECOVER_SERVERFAIL;
         do {
                time_t  now;
                time( &now );
                if (now >= end_time)
                {
                    rec_api_log_csm( SUB_CSM_RPC_FAIL,
                                "timeout connecting to restore engine" );
                    return EP_RB_RECOVER_SERVERFAIL;
                }

                sleep( 1 );     /* give restore engine time to get going */
                retval = csc_connect_to_rpc_service(
                                        (unsigned char *)hostname,
                                        re_if_spec,
                                        RE_CLIENT_GROUP,
                                        &handlePtr -> re_binding_handle,
                                        &status);
                if ((status == error_status_ok) && (retval != 0))
                        api_status = E_SUCCESS;
        } while (api_status != E_SUCCESS);

        if (api_status == E_SUCCESS)
        {
                re_handle = handlePtr -> re_binding_handle;
#ifdef DEBUG
/*      increase rpc timeout during debugging */
                rpc_timeout.tv_sec = RPC_TIMEOUT;
                rpc_timeout.tv_usec = 0;
                clnt_control( re_handle, CLSET_TIMEOUT, (
                                                    char *)&rpc_timeout );
#endif
                re_init_args.username = human_uidname;
                set_rpc_obj( re_initialize, &re_init_args.RPCobjID );
                re_init_result = re_initialize_1( &re_init_args, re_handle );
                if (!re_init_result) {
                        api_status = EP_RB_RECOVER_RPC_FAIL;
                        rec_api_log_csm( SUB_CSM_RPC_FAIL,
                            "failure communicating with restore engine" );
                }
                else {
                        api_status = re_init_result->status;
                        /* release RPC result struct: */
                        xdr_free( xdr_RE_status_result, (
                                                    char *)re_init_result);
                }
        }

        else
            rec_api_log_csm( SUB_CSM_RPC_FAIL,
                            "failure connecting to restore engine" );

        if (
          api_status == E_SUCCESS)     /* return rest eng handle on success */
```

```
            *svrHdl = (serverHandle)re_handle;

        return( api_status );

}       /* End of EDMRST_Initialize() */
```

```
/*******************************************************************
 * Ping:
 *
 *  This function allows a ping to be issued in order to keep the
 *  engine alive and running so that the engine will not time out.
 *
 * Parameters:
 *
 *  svrHdl (I) - A pointer to this user's client handle for the
 *                 Restore Engine (server) connection.
 *
 *******************************************************************/
eerrno_ty EDMRST_Ping( serverHandle svrHdl )
{
  eerrno_ty      api_status = E_SUCCESS;
  RE_null_args   re_ping_args;
  RE_status_result      *re_ping_result = NULL;


  if ( NULL == svrHdl || NULL == handlePtr
       || svrHdl != handlePtr->re_binding_handle )
    {
      return( EP_RB_RECOVER_BAD_ARGS );
    }

  set_rpc_obj( re_ping, &re_ping_args.RPCobjID );
  re_ping_result = re_ping_1( &re_ping_args, svrHdl );
  if (NULL == re_ping_result) {
    api_status = EP_RB_RECOVER_RPC_FAIL;
    rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
  }
  else {
    api_status = re_ping_result->status;
    /* release RPC result struct: */
    xdr_free( xdr_RE_status_result, (char *)re_ping_result);
  }

}



/*******************************************************************
 * EDMRST_Finish
 *
 *    Function Description:
 *
 *  This function terminates a restoral session,
                                            but only during the browse and
 *  mark phase. It will be rejected if a restore is currently being executed.
 *  This routine will clean up any local memory used in the session and will
 *  disconnect from the Restore Engine.  After calling this function,
 *  EDMRST_Initialize MUST be called before calling any other functions in
                                                                        this
 *  API.
 *
 * Parameters:
 *
 *  svrHdl (I) - A pointer to this user's client handle for the
 *                Restore Engine (server) connection.
 *
 *    Return Codes:
 *        EP_RB_RECOVER_BAD_ARGS
 *        EP_RB_RECOVER_RPC_FAIL
 *        EP_RB_RECOVER_INVALOP
 *        EP_RB_RECOVER_SERVERFAIL
 *
 */

eerrno_ty
EDMRST_Finish( serverHandle svrHdl )
{
        eerrno_ty       api_status = E_SUCCESS;
        RE_null_args    re_finish_args;
        RE_status_result        *re_finish_result = NULL;
        int             csc_status;

        if ( NULL == svrHdl || NULL == handlePtr
          || svrHdl != handlePtr->re_binding_handle )
        {
                return( EP_RB_RECOVER_BAD_ARGS );
        }

        set_rpc_obj( re_finish, &re_finish_args.RPCobjID );
        re_finish_result = re_finish_1( &re_finish_args, svrHdl );
        if (!re_finish_result) {
                api_status = EP_RB_RECOVER_RPC_FAIL;
                rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
        }
        else {
                api_status = re_finish_result->status;
                /* release RPC result struct: */
                xdr_free( xdr_RE_status_result, (char *)re_finish_result);
        }

        rec_api_log_end();      /* write last log and close the log file. */


        return( api_status );

}       /* EDMRST_Finish */
```

```
/*
 ** Copyright 1996,1997 EMC Corporation
 */


/* EDMDispatchService.c
 *
 *
 * Mission Statement:  RPC entry points.
 *
 * Primary Data Acted On:
 *
 * Compile-Time Options:
 *
 * Basic idea here:
 */

#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ "
                            "$Revision: 1.0 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

#include <esl/c_portable.h>
#include <esl/inout.h>

#include <logging/logging.h>
#include <csc/cscomm.h>

#include <restore/csc_EDMDispatch.h>
#include <restore/dispatch_daemon.h>

#include <EDMDispatchLog.h>
#include <EDMDispatchSession.h>

/*
 * These are all the rpc entry points for the dispatch daemon.
 * The dispatch daemon is multi-threaded and it is the main thread
 * which handles all incoming RPC. ONC RPC is single threaded
 * so each call blocks other RPC calls. This provides us some
 * safety in the way we handle our data and limits our exposure
 * to unexpected multithreading problems.
 */
static void FreeSessionInfo(SessionInfo *);

/*****************************************************************************
**
** Routine:  dd_initialize_1
**
** Inputs:   DD_initialize_args * - args for the restore initialize call
**
** Outputs:  None
**
** Return Codes:
**           DD_initialize_result * - result of init function call
**
** Purpose:  Function to create a restore session.
**
** Intended caller:  Internal Only.
******************************************************************************
*/

DD_initialize_result *
dd_initialize_1_svc(IN DD_initialize_args *arg, IN struct svc_req *req )
{
    static DD_initialize_result argzz;
```

```
    InitializeSession(arg, req, &argzz);

    return &argzz;
}

/*****************************************************************************
**
** Routine:  dd_getservicestatus_1
**
** Inputs:   DD_getservicestatus_args * - args for the getservicestatus call
**
** Outputs:  None
**
** Return Codes:
**           DD_getservicestatus_result * - result of status function call
**
** Purpose:  Function to poll for status on a session.
**
** Intended caller:  Internal Only.
******************************************************************************
*/

DD_getservicestatus_result *
dd_getservicestatus_1_svc(
                       IN DD_getservicestatus_args *arg, IN struct svc_req *req )
{
    static DD_getservicestatus_result argzz;

    GetDispatchStatus(arg, &argzz);

    return &argzz;
}

/*****************************************************************************
**
** Routine:  dd_getsessioninfo_1
**
** Inputs:   DD_getservicestatus_args * - args for the getsessioninfo call
**
** Outputs:  None
**
** Return Codes:
**           SessionBlock * - result of session info call
**
** Purpose:  Function to get information on all sessions.
**
** Intended caller:  Internal Only.
******************************************************************************
*/

SessionBlock *
dd_getsessioninfo_1_svc(
                       IN DD_getservicestatus_args *arg, IN struct svc_req *req )
{
    static SessionBlock argzz;
    static boolean_ty first = TRUE;

    if (first)
    {
        memset(&argzz, 0, sizeof(argzz));
        first = FALSE;
    }
    else
    {
        FreeSessionInfo(argzz.sess);
        argzz.sess = NULL;
    }

    GetDispatchInfo(arg, &argzz);

    return &argzz;
}

/*****************************************************************************
**
** Routine:  FreeSessionInfo
**
** Inputs:   SessionInfo * - arg to free
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Function to free all SessionInfo structures in a list.
**
** Intended caller:  Internal Only.
*****************************************************************************
                                                                           */

static void FreeSessionInfo(SessionInfo *sess)
{
    if (sess == NULL)
        return;

    if (sess -> next != NULL)
        FreeSessionInfo(sess -> next);

    free(sess);
}
```

```
/*
** Copyright 1996,1999 EMC Corporation
*/


/*
** EDMDispatchSession.cc
**
** Mission Statement: This is where all session management occurs.
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                     USE_SUNRPC - Compile source with sunrpc support.  If
**                                  not set, assume DCE support.
**
** Basic idea here: Module for session management
*/

/*
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <pthread.h>
#include <memory.h>
#include <sys/time.h>
#include <sys/types.h>
#include <syslog.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>

#include <csc/cscomm.h>
#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol_client.h>
#include <EDMSession.h>
#include <EDMReturnMessageApi.h>
#include <EDMDDHandleMgrApi.h>
#include <EDMDispatchSession.h>
#include <EDMDispatchConfig.h>
#include <EDMDDcr_rstsvc.h>

#include <EDMDispatchLog.h>

static RWBinaryTree      G_sessionTree;

static pthread_mutex_t    G_sessionTreeMtx = PTHREAD_MUTEX_INITIALIZER;
extern ELinkHandlePtr_ty  ELinkHandle;

static int maxDisconnectTime = SECONDS_PER_HOUR; // one hour
```

```
/*****************************************************************************
**
** Routine:  LockSessionMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the session mutex.
**
*****************************************************************************
*/

static void
LockSessionMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_sessionTreeMtx, NULL);
    }

    pthread_mutex_lock(&G_sessionTreeMtx);
}

/*****************************************************************************
**
** Routine:  UnlockSessionMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for the session tree object
**
*****************************************************************************
*/

static void
UnlockSessionMutex()
{
    pthread_mutex_unlock(&G_sessionTreeMtx);
}

/*****************************************************************************
**
** Routine:  InitializeSession
**
** Inputs:   DD_initialize_args *arg - args sent via RPC for starting session
**           struct svc_req *req - the request block from RPC
**
** Outputs:  DD_initialize_result *res - the result structure which tells
                                                                      whether
**                      operation succeeded or failed.
**
** Return Codes:
**           None
```

```
**
** Purpose:  Initialize a session for the GUI.
**
*****************************************************************************
*/

void
InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
                  OUT DD_initialize_result *res)
{
    EDMSession  *session;
    EDMSession  *ret;
    pthread_t   id;
    time_t       t;

    if (arg == NULL || req == NULL || res == NULL)
    {
        return;
    }

    t = time(NULL);

    session = new EDMSession();

    if (session == NULL)
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        return;
    }

    session -> initSession();

    session -> setStartTime(t);

    session -> setOperationType(arg -> service);

    session -> setStatus(DD_SERVICE_STARTING);

    if (arg -> username  != NULL && arg -> hostname != NULL)
    {
      switch(arg -> service)
        {
           // code is commented out because we do not
           // want to read tthe config for permission information
           // at this time, it is a waste of cycles
#if 0
             case DD_SERVICE_RESTORE : boolean_ty allowed;

                                  allowed = DispatchCheckRestorePermission(
                                          arg->hostname, arg -> username);

                                  if (!allowed)
                                 {
                                      res -> status = DD_SERVICE_FAILURE_PERMS;
                                     delete session;
                                      return;
                                  }

                                  break;
#endif
        default: // Add some error messsage for unknown service
                                   break;
        };
    }
    else
    {
```

```
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        delete session;
        return;
    }

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.insert((RWCollectable *) session);

    UnlockSessionMutex();

    if (ret == NULL)
    {
        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        delete session;
        return;
    }

    session -> getSessionID(&res -> service_handle);

    // Call Steve's thread
    pthread_create(&id, NULL, &DDRSTsvc_init, (void *) session);

    session -> setThreadID(id);

    return;
}

/*****************************************************************************
**
** Routine:  SendPingMessagesToSession
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Queue up all the ping messages to the sessions. If they don't
**           respond they should be considered dead.
**
*****************************************************************************
*/

void
SendPingMessagesToSession()
{
    EDMSession *sess;

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                            G_sessionTree);

    while ( sessionIterator != NULL &&
          (sess = (EDMSession*) (*sessionIterator)()) != NULL )
    {
        DD_client_session_id sid;
        rpc_binding_handle_t *cscb = NULL;
        int                     status;
        int                     ret;

        if (sess -> getStatus() != DD_SERVICE_RUNNING)
            continue;
        sess -> getSessionID(&sid);

        ret = GetCSCHandle(&sid, &cscb, &status);

        if (ret != 0 || cscb == NULL || *cscb == NULL)
            continue;

        PushResponseMessage(dp_ping_request, sid, cscb, &status);
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    UnlockSessionMutex();
}

/******************************************************************************
**
** Routine:  UpdateSessionLastReceived
**
** Inputs:   DD_client_session_id *sessID - session that sent us something
**
** Outputs:  None
**
** Return Codes:
**           0 on success and non-zero otherwise
**
** Purpose:  Update the specified session with the lastest received message
**           time.
**
*******************************************************************************
*/

int
UpdateSessionLastReceived(DD_client_session_id *sessID)
{
    time_t    last = time(NULL);
    EDMSession  *session;
    EDMSession  *ret;

    session = new EDMSession();

    if (session == NULL)
    {
        EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        return -1;
    }

    session -> setSessionID(sessID);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

    UnlockSessionMutex();

    delete session;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
```

```
                            "Failure to update session %ld:%ld received time",
                            sessID -> high, sessID -> low);
        return -1;
    }

    ret -> setLastReceived(last);

    return 0;
}

/******************************************************************************
**
** Routine:  UpdateSessionLastSent
**
** Inputs:   DD_client_session_id *sessID - session that sent us something
**
** Outputs:  None
**
** Return Codes:
**           0 on success and non-zero otherwise
**
** Purpose:  Update the specified session with the lastest sent message
**           time.
**
*******************************************************************************
*/

int
UpdateSessionLastSent(DD_client_session_id *sessID)
{
    time_t    last = time(NULL);
    EDMSession  *session;
    EDMSession  *ret;

    session = new EDMSession();

    if (session == NULL)
    {
        EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        return -1;
    }

    session -> setSessionID(sessID);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);

    UnlockSessionMutex();

    delete session;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                            "Failure to update session %ld:%ld sent time",
                            sessID -> high, sessID -> low);
        return -1;
    }

    ret -> setLastSent(last);

    return 0;
}
```

```
/*****************************************************************************
**
** Routine:  CheckDispatchSessions
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Look for dead sessions and kill them off
**
*****************************************************************************
*/

void
CheckDispatchSessions()
{
    EDMSession   *sess;
    int          status = 0;
    int          ret = 0;
    time_t       currTime;
    RWBinaryTree reaperTree;

    currTime = time(NULL);

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                        G_sessionTree);

    while ( sessionIterator != NULL &&
          (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {

        if ( (sess->getLastReceived(
         ) <= currTime - maxDisconnectTime && sess->getLastReceived() != 0) ||
            (sess->getStartTime() <= currTime - maxDisconnectTime &&
            (sess -> getStatus(
                  ) == DD_SERVICE_FAILURE_NONEXEC || sess -> getStatus(
                                    ) == DD_SERVICE_FAILURE_EXEC ||
             sess -> getStatus() == DD_SERVICE_FAILURE_PERMS)) )
        {
            // Insert it into the reaper tree
            (void) reaperTree.insert(sess);
        }
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    UnlockSessionMutex();

    // If the reaper tree has something in it then use those entries to remove
    // things from the query tree.
    if (reaperTree.entries() > 0)
    {
        sessionIterator = new RWBinaryTreeIterator(reaperTree);

        while ( sessionIterator != NULL &&
              (sess = (EDMSession*) (*sessionIterator)()) != NULL ) {
            DD_client_session_id  sessID;
```

```
            sess -> getSessionID(&sessID);

            ret = removeSession(&sessID, &status);

            if (ret != 0)
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                                   "Failure to remove session %ld:%ld",
                                   sessID.high, sessID.low);
                continue;
            }
            else
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO, 0, 0,
                          "Removing session %ld:%ld,
                            Haven't recieved anything since %ld. Current %ld",
                          sessID.high, sessID.low, sess -> getLastReceived(),
                          currTime - maxDisconnectTime);
            }

            ret = deleteHandleSet(&sessID, &ELinkHandle, &status);

            if (ret != 0)
            {
                EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                                   "Failure to delete handles for session
                                                                  %ld:%ld",

                                   sessID.high, sessID.low);
            }
        }

        // through with iterator
        if (sessionIterator != NULL)
        {
            delete sessionIterator;
        }

        reaperTree.clear();
    }
}

/*****************************************************************************
**
** Routine:  DrainSessionDescriptors
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Drain whatever data is on stdout and stderr for sessions.
**
*****************************************************************************
*/

void
DrainSessionDescriptors()
{
    int     hout = 0, herr = 0, status = 0;
    int     selret = 0;
    int     i = 0;
    char     buff[1024];
    struct timeval timetowait = {
            1, 0
            };
    fd_set  stdoutSet;
    fd_set  stderrSet;

    getStdoutSet(&stdoutSet, &hout, &status);

    if ( (selret = select(
                        hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0)
    {
        for (; i < hout+1; i++)
        {
            if (FD_ISSET(i, &stdoutSet))
            {
                while (read(i, buff, 1024) > 0);
            }
        }
    }

    getStderrSet(&stderrSet, &herr, &status);

    if ( (selret = select(
                        herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0)
    {
        for (i = 0; i < herr+1; i++)
        {
            if (FD_ISSET(i, &stderrSet))
            {
                while (read(i, buff, 1024) > 0);
            }
        }
    }
}

/*****************************************************************************
**
** Routine:  GetSessionStatus
**
** Inputs:   DD_client_session_id *ssid - session ID to check the status of
**
** Outputs:  int *status - status of the function call
**           int *s_status - session status
**
** Return Codes:
**           0 if successful and non-zero otherwise
**
** Purpose:  Get status on the session.
**
*****************************************************************************
*/
int
GetSessionStatus(DD_client_session_id *ssid, int *s_status, int *status)
{
    EDMSession  *sess;
    EDMSession  *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (ssid == NULL || s_status == NULL)
    {
        *status = SESSION_BAD_ARGS;

        return -1;
```

```
    }

    sess = new EDMSession();

    if (sess == NULL)
    {
        EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        *status = SESSION_NO_MEMORY;

        return -1;
    }

    sess -> setSessionID(ssid);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

    UnlockSessionMutex();

    delete sess;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                         "Failure to lookup session %ld:%ld",
                         ssid -> high, ssid -> low);
        *status = SESSION_LOOKUP_FAILED;
        return -1;
    }

    *s_status = ret -> getStatus();

    return 0;
}

/*****************************************************************************
**
** Routine:  GetDispatchStatus
**
** Inputs:   DD_getservicestatus_args *arg - session ID to check the status of
**
** Outputs:  DD_getservicestatus_result *res - the result structure which
                                                                        tells
**                     whether operation succeeded or failed.
**
** Return Codes:
**           None
**
** Purpose:  Get status on the starting session.
**
*****************************************************************************
*/

void
GetDispatchStatus(IN DD_getservicestatus_args *arg,
                  OUT DD_getservicestatus_result *res)
{
    EDMSession  *sess;
    EDMSession  *ret;
    static char buff[CONNECT_HANDLE_SIZE];

    sess = new EDMSession();
    if (sess == NULL)
    { // Give an error
        EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                            "Failure to create a session block");
        return;
    }

    sess -> setSessionID(&arg -> service_handle);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);

    UnlockSessionMutex();

    delete sess;

    if (ret == NULL)
    {
        EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                          "Failure to lookup session %ld:%ld",
                          arg -> service_handle.high,
                                             arg -> service_handle.low);

        res -> status = DD_SERVICE_FAILURE_NONEXEC;
        return;
    }

    res -> status = ret -> getStatus();

    memset(buff, 0, sizeof(buff));

    if (res -> status == DD_SERVICE_RUNNING)
    {
        res -> handle.handle_val = (char *) ret -> getConnectionHandle();
        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
    }
    else
    {
        res -> handle.handle_val = (char *) buff;
        res -> handle.handle_len = CONNECT_HANDLE_SIZE;
    }
}

/***************************************************************************
**
** Routine:  GetDispatchInfo
**
** Inputs:   DD_getservicestatus_args *arg - session ID to check the status of
**
** Outputs:  SessionBlock *res - the information regarding the specified
                                                                      session
**
** Return Codes:
**           None
**
** Purpose:  Get status on all the sessions.
**
****************************************************************************
*/

void
GetDispatchInfo(IN DD_getservicestatus_args *arg,
                OUT SessionBlock *res)
{
```

```
    EDMSession   *sess;
    EDMSession   *ret;
    SessionInfo  *sinfo, *slast;
    static char buff[CONNECT_HANDLE_SIZE];

    LockSessionMutex();

    if (arg -> service_handle.high != 0 && arg -> service_handle.low != 0)
    {
        // Looking for a single session. Do a find.
        sess = new EDMSession();

        if (sess == NULL)
        { // Give an error
            EDMDispatch_logent(
                            __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                              "Failure to create a session block");

            UnlockSessionMutex();
            return;
        }

        sess -> setSessionID(&arg -> service_handle);

        ret = (EDMSession *) G_sessionTree.find(sess);

        delete sess;

        if (ret == NULL)
        {
            EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                              "Failure to lookup session %ld:%ld",
                          arg -> service_handle.high,
                                             arg -> service_handle.low);
            UnlockSessionMutex();
            return;
        }

        res -> totalsessions = 1;

        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

        if (res -> sess == NULL)
        {
            EDMDispatch_logent(
                            __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                              "Failure to allocate session info block");
            UnlockSessionMutex();
            return;
        }

        sinfo = res -> sess;

        ret -> getSessionID(&sinfo -> service_handle);
        sinfo -> status = ret -> getStatus();
        sinfo -> jobstarttime = ret -> getStartTime();
        sinfo -> operation_type = ret -> getOperationType();
        sinfo -> lastSent = ret -> getLastSent();
        sinfo -> lastReceived = ret -> getLastReceived();
    }
    else
    {
        res -> totalsessions = 0;

        res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));
        if (res -> sess == NULL)
        {
            EDMDispatch_logent(
                           __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                              "Failure to allocate session info block");
            UnlockSessionMutex();
            return;
        }

        sinfo = res -> sess;

        RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                          G_sessionTree);

        boolean_ty addnext = FALSE;

        while ( sessionIterator != NULL && (ret = (EDMSession*) (
                                             *sessionIterator)()) != NULL )
        {
            int                 status;

            if (addnext)
            {
                sinfo -> next = (SessionInfo *) calloc(1, sizeof(SessionInfo));

                if (sinfo -> next == NULL)
                {
                    break;
                }

                sinfo = sinfo -> next;
            }

            ret -> getSessionID(&sinfo -> service_handle);
            sinfo -> status = ret -> getStatus();
            sinfo -> jobstarttime = ret -> getStartTime();
            sinfo -> operation_type = ret -> getOperationType();
            sinfo -> lastSent = ret -> getLastSent();
            sinfo -> lastReceived = ret -> getLastReceived();

            getHandleSet(&sinfo -> service_handle, &sinfo -> outhandle,
                         &sinfo -> errhandle, &status);

            res -> totalsessions++;

            sinfo -> next = NULL;
            addnext = TRUE;
        }

        // through with iterator
        if (sessionIterator != NULL)
        {
            delete sessionIterator;
        }

    }

    UnlockSessionMutex();
}

/******************************************************************************
**
** Routine:  removeSession
**
** Inputs:
```

```
**
** Outputs:
**
** Return Codes:
**          None
**
** Purpose: Remove the active session object between the GUI and the Service.
**
*******************************************************************************
*/

int
removeSession(IN DD_client_session_id *sess_id,
              OUT int *status)
{
    EDMSession *sess;
    EDMSession *ret;

    if (status == NULL)
    {
        return -1;
    }

    if (sess_id == NULL)
    {
        *status = SESSION_BAD_ARGS;
        return -1;
    }

    *status = 0;
    if (G_sessionTree.isEmpty())
    {
        EDMDispatch_logent(
                          __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
                          "No sessions in list.
                                     Can't remove session <%ld:%ld>",
                          sess_id -> high, sess_id -> low);
        *status = SESSION_LIST_EMPTY;
        return -1;
    }

    sess = new EDMSession();

    if (sess == NULL)
    {
        EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                          "Failure to create a session block");
        *status = SESSION_NO_MEMORY;
        return -1;
    }

    sess -> setSessionID(sess_id);

    LockSessionMutex();

    ret = (EDMSession *) G_sessionTree.remove(sess);

    UnlockSessionMutex();

    if (ret == NULL)
    {
        EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                          "Failure to remove session %ld:%ld",
                          sess_id -> high, sess_id -> low);
        delete sess;
        *status = SESSION_LOOKUP_FAILED;
        return -1;
    }

    delete ret;
    delete sess;

    return 0;
}
```

```
/*
** ===========================================================================
** Copyright 1996,1997 EMC Corporation
** ===========================================================================
*/
/*
** ===========================================================================
** DDRSTsvc_init.c
**
** Mission Statement:
**
**
**
**
** Primary Data Acted On:
**
** Compile-Time Options:
**
**                      USE_SUNRPC - Compile source with sunrpc support.  If
**                                   not set, assume DCE support.
**
** Basic idea here:
** ===========================================================================
*/
/*
** ===========================================================================
** The following provides an RCS id in the binary that can be located
** with the what(1) utility.  The intent is to keep this short.
** ===========================================================================
*/
#if !defined(lint)
static char     RCS_id [] = "@(#)$RCSfile: EDMccr.c,v $ "
                            "$Revision: 1.23 $ "
                            "$Date: 1997/02/06 20:49:15 $" ;
#endif

/* #define _POSIX_SOURCE   unable to compile with this define set */
/* #define _XOPEN_SOURCE   unable to compile with this define set */

#include <sys/types.h>
#include <sys/utsname.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <esl/c_portable.h>
#include <esl/ep_xopen.h>
#include <esl/inout.h>

#include <string.h>
#include <stdlib.h>
#include <pthread.h>

// Rogue Wave includes
#include <rw/collect.h>
#include <rw/rwfile.h>
#include <rw/vstream.h>
#include <rw/bintree.h>

#include <csc/cscomm.h>
#include <edmlink/edmlink_api.h>

#ifdef __cplusplus
extern "C" {
#endif
```

```
#include <restore/dispatch_daemon.h>
#include <restore/dispatch_protocol.h>
#include <restore/RestoreObjectID.h>
#include <restore/csc_Dispatch_Protocol_Service.h>
#include <restore/dispatch_protocol_service.h>
#include <restore/dispatch_protocol_client.h>
#include <dpService.h>

#ifdef __cplusplus
}
#endif

#include <logging/logging.h>
#include <EDMDispatchLog.h>
#include <EDMDDHandle.h>
#include <EDMDDHandleMgrApi.h>
#include <EDMSession.h>
#include <EDMccr.h>
#include <EDMutils.h>
#include <EDMDD_ddp.h>
#include <EDMDDcr_rstsvc.h>

pthread_cond_t cscPortRdy_cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t cscPortRdy_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t G_serviceMtx;

static boolean32 print_error = TRUE;

/* Prototypes */
int edmrst_send_chndl_to_private_svc(int);
int edmrst_create_ddp_client_connection(
                                  int,rpc_binding_handle_t **,EDMSession  *);
int edmrst_send_uid_to_private_svc(int, EDMSession *);

/* Dispatch Protocol ifspec */
static rpc_if_handle_t DispatchDaemon_ifspec;
ELinkHandlePtr_ty  ELinkHandle;     /* Handle for svc object */

/*****************************************************************************
**
** Routine:  LockSvcMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Lock the mutex for the service execution
**
******************************************************************************
*/

static void
LockSvcMutex()
{
    static boolean_ty first = TRUE;

    if (first == TRUE)
    {
        first = FALSE;
        pthread_mutex_init(&G_serviceMtx, NULL);
    }
    pthread_mutex_lock(&G_serviceMtx);
}

/*****************************************************************************
**
** Routine:  UnlockSvcMutex
**
** Inputs:   None
**
** Outputs:  None
**
** Return Codes:
**           None
**
** Purpose:  Unlock the mutex for service execution
**
*****************************************************************************
*/

static void
UnlockSvcMutex()
{
    pthread_mutex_unlock(&G_serviceMtx);
}

void *
DDRSTsvc_init(void *pSessObj)
{
int                        lrc;              /* Local Return Code             */
int                         fd1;
int                         fd2;
int                        status;
rpc_binding_handle_t * bh=NULL;
EDMSession *                p_so;
// EDMDDHandle *                    pEHandleObj;
DD_client_session_id    sID;
ELinkShellObjPtr_ty     ShellHandle;
unsigned char           *svc_rpc_h=NULL; /* X-Service RPC Handle */
ELinkTargetObjPtr_ty    TargetObjPtr;    /* Target object all functions    */
ELinkUserIdObjPtr_ty    UserIdObjPtr;    /* UserId object copy & shell     */
ELinkCmdObjPtr_ty       CmdObjPtr;       /* Shell command shell only       */
unsigned long           options = 0;     /* For ELinkNewServiceLaunchObj   */


  //
  // launch one service at a time.
  //
  LockSvcMutex();
  pthread_mutex_lock( &cscPortRdy_mutex );

  //
  // Check to see that the EDMLINK handle didn't get trashed
  //
  if (ELinkHandle == NULL)
  {
      UnlockSvcMutex();
      pthread_mutex_unlock( &cscPortRdy_mutex);
      pthread_exit( NULL );
  }

  //
  // Cast the input argument to its object type.
  //
  p_so = (EDMSession*)pSessObj;

  //
```

```
  // Construct EDM-Link target object so that EDM-Link will know what
  // system we want to talk to.
  //

  TargetObjPtr = ELinkNewTargetObj( ELinkHandle,
                                     "localhost" );

  //
  // EDM-Link should have called our callback DOMIELinkCallback which
  // should have loaded DOMIHandle->ErrorBlock, so all we have to do
  // now, is return.
  //
  if ( NULL == TargetObjPtr )
  {
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_mutex_unlock( &cscPortRdy_mutex);
      pthread_exit( NULL );
  }

  //
  // Construct EDM-Link user object.  We always want to run as root on the
  // target.  We are starting a private service that will run on an EDM.
  // We know that we will be starting via the EDM-Link daemon and we can
  // always start using the root id.  Also, this will be a service, it needs
  // to run as root and will have some intelligence in protecting itself in
  // that there a limited things that it can do and the caller of the API
  // will control what can be done.
  //
  UserIdObjPtr = ELinkNewUserIdObj( ELinkHandle,
                                     TargetObjPtr,
                                     NULL,
                                     NULL );


  //
  // EDM-Link should have called our callback DOMIELinkCallback which
  // should have loaded DOMIHandle->ErrorBlock, so all we have to do
  // now, is return.
  //
  if ( NULL == UserIdObjPtr )
  {
      (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_mutex_unlock( &cscPortRdy_mutex);
      pthread_exit( NULL );
  }

  //
  // Utilize the EDM-Link service launcher to physically startup the
  // domain private service.  By convention, all private services can
  // be found in /usr/epoch/service and have a suffix of pd.  The domain
  // private service is: /usr/epoch/service/domainpd.
  //

  if (IsDebugOn())
      options |= ELINK_SERVICE_DEBUG;
                                  /* if we are debug, so will be R.Eng */

  CmdObjPtr = ELinkNewServiceLaunchObj( ELinkHandle,
                                         TargetObjPtr,
                                         "edmrestoreeng",
                                                /* Domain private service */
                                         options );
  //
  // EDM-Link should have called our callback DOMIELinkCallback which
```

```
    // should have loaded DOMIHandle->ErrorBlock, so all we have to do
    // now, is return.
    //
    if ( NULL == CmdObjPtr )
    {
        (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
        (void) ELinkDestroyObj( ELinkHandle, UserIdObjPtr );
        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
        UnlockSvcMutex();
        pthread_mutex_unlock( &cscPortRdy_mutex);
        pthread_exit( NULL );
    }

   //
   // Fire up Private Service via EDM-Link API ELinkPrivateSvc.  This
   // physically starts the private service running.
   //
   lrc = ELinkPrivateSvc ( ELinkHandle,
                           TargetObjPtr,
                           UserIdObjPtr,
                           CmdObjPtr,
                           &fd1,
                           &fd2,
                           &ShellHandle );
   if ( -1 == lrc )
     {
       (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
       (void) ELinkDestroyObj( ELinkHandle, UserIdObjPtr );
       (void) ELinkDestroyObj( ELinkHandle, CmdObjPtr );
       EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_PRIVATE_SVC_FAILURE,
                         0,"ELinkPrivateSvc() failure");
       p_so -> setStatus(DD_SERVICE_FAILURE_EXEC);
       UnlockSvcMutex();
       pthread_mutex_unlock( &cscPortRdy_mutex);
       pthread_exit( NULL );
     }

   (void) ELinkDestroyObj( ELinkHandle, TargetObjPtr );
   (void) ELinkDestroyObj( ELinkHandle, UserIdObjPtr );
   (void) ELinkDestroyObj( ELinkHandle, CmdObjPtr );

   //
   // Extract the csc handle from the shell object. This handle
   // is the restore service (restore API) rpc handle.
   //
   svc_rpc_h = (unsigned char*) calloc(1,CONNECT_HANDLE_SIZE);
   if (svc_rpc_h == NULL)
   {
       EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_NO_MEMORY,
                         0,"calloc() failure");
       p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
       UnlockSvcMutex();
       pthread_mutex_unlock( &cscPortRdy_mutex);
       pthread_exit( NULL );
   }
   lrc = ELinkGetConnectHandle( ELinkHandle,
                                ShellHandle,
                                svc_rpc_h );
   if ( 0 != lrc )
     {
       (void) free(svc_rpc_h);
       EDMDispatch_logent(
                   __FILE__,__LINE__,LOG_ERR,DDP_GET_CONNECT_HANDLE_FAILURE,
               0,"edmrst_get_client_rpc_handle() failure");
       p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
       UnlockSvcMutex();
```

```
        pthread_mutex_unlock( &cscPortRdy_mutex);
        pthread_exit( NULL );
     }

    p_so -> setConnectionHandle((void *)svc_rpc_h);

    p_so -> getSessionID(&sID);              // Get Unique Session id

    //
    // Issue message telling of Dispatch Daemon RDR port number.
    //

    if (IsDebugOn())
    {
       EDMDispatch_logent(__FILE__,__LINE__,LOG_INFO,DDP_PORT_NUMBERS,
                         0,"PORT_INFO DispatchDaemon_ifspec(DDCCR) port#: %d",
                         DispatchDaemon_ifspec.portnum);
    }

    //
    // Unlock Port Rdy mutex so the Reader can listen.
    //
    pthread_mutex_unlock( &cscPortRdy_mutex);

    //
    // Tell the Dispatch Daemon Protocol Reader Thread to Listen.
    //
    pthread_cond_signal(&cscPortRdy_cv);

    //
    // Inform the restore svc of dispatch protocol details (port etc ...)
    //
    lrc = edmrst_send_chndl_to_private_svc(fd1);
    if ( 0 != lrc )
    {
        (void) free(svc_rpc_h);
        EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_CHANNEL_SEND_FAILURE,
                          0,"edmrst_send_chndl_to_private_svc() failure");
        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
        UnlockSvcMutex();
        pthread_exit( NULL );
    }

    //
    // Send the Unique Session Id value.
    //
    lrc = edmrst_send_uid_to_private_svc(fd1,p_so);
    if ( 0 != lrc )
    {
        (void) free(svc_rpc_h);
        EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_SEND_UID_FAILURE,
                          0,"edmrst_send_uid_to_private_svc() failure");
        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
        UnlockSvcMutex();
        pthread_exit( NULL );
    }

    //
    // Create the CCW service handle so we can respond to messages.
    //
    lrc = edmrst_create_ddp_client_connection(fd1, &bh, p_so);
    if ( 0 != lrc )
    {
        (void) free(svc_rpc_h);
        EDMDispatch_logent(
                    __FILE__,__LINE__,LOG_ERR,DDP_CREATE_CLIENT_CONNECTION,
                              0,"edmrst_create_ddp_client_connection() failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_exit( NULL );
   }

   //
   // Insert handle object into Global list.
   //
   lrc = newHandleSet( &sID,
                       fd1,
                       fd2,
                       bh,
                      &ShellHandle,
                       &status );
   if ( 0 != lrc )
   {
      (void) free(svc_rpc_h);
      EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_HANDLE_INSERTION_ERROR,
                         status,"newHandleSet() failure");
      p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
      UnlockSvcMutex();
      pthread_exit( NULL );
   }

   //
   // Let's clean up and set the status to RUNNING.
   //
   p_so -> setStatus(DD_SERVICE_RUNNING);
   UnlockSvcMutex();
   pthread_exit( NULL );
  return( NULL );
}

/*
** ============================================================================
**  Function:    edmrst_send_chndl_to_private_svc()
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
**               <0 Read less than exspected
** ============================================================================
*/
int
edmrst_send_chndl_to_private_svc(int pipeToSvc)
{
auto int lrc=0;
auto unsigned char *p_client_h=NULL;

//
//  Isolate the connection handle from the server 'if_spec'.
//  The IP/PORT are part of the created if_spec structure.
//
p_client_h = DispatchDaemon_ifspec.connect_handle_p;

//
// Write the handle to the service so it can contact me
//
lrc = edmrst_WrChannel(pipeToSvc,
                       p_client_h,
                       CONNECT_HANDLE_SIZE);
if ( CONNECT_HANDLE_SIZE != lrc )
  {
```

```
    (void) free(p_client_h);
    EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_WRITE_CHANNEL,
                            0,"edmrst_WrChannel() Failure");
    return(-1);
  }

return(0);
}

/*
** ============================================================================
**  Function:    edmrst_send_uid_to_private_svc()
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
**               <0 Read less than exspected
** ============================================================================
*/
int
edmrst_send_uid_to_private_svc(int pipeToSvc,
                                 EDMSession  *pSessionObj)
{
auto int lrc=0;
auto DD_client_session_id uid;

//
// Write the handle to the service so it can contact me
//
pSessionObj -> getSessionID(&uid);
lrc = edmrst_WrChannel(pipeToSvc,
                       (void*)&uid,
                       sizeof(DD_client_session_id));
if ( (sizeof(DD_client_session_id)) != lrc )
  {
    EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_WRITE_CHANNEL,
                            0,"edmrst_WrChannel() Failure");
    return(-1);
  }

return(0);
}
/*
** ============================================================================
**  Function:    edmrst_create_ddp_client_connection()
**  Description:
**
**
**
**  Returns:      0 Sucessful
**               -1 Read Failure
**               <0 Read less than exspected
** ============================================================================
*/
int
edmrst_create_ddp_client_connection(int pipeToSvc,
                                    rpc_binding_handle_t **bh,
                                     EDMSession  *p_so )
{
  int lrc;
  unsigned char *p_restore_service=NULL;
  error_status_t status;
  rpc_if_handle_t *p_psvc_ifspec=NULL;
  rpc_binding_handle_t *psvc_h=NULL;
  //
  // We now need to get the details from the restore service on
  // how to connect from the dispatch daemon ccw to the restore
  // service ccr. At this point, the restore service will be send -
  // ing the restore service ccr handle information. The port / ip
  // are the key information needed to create the ddp ccw handle.
  //
  lrc = edmrst_get_client_handle( pipeToSvc,&p_restore_service );
  if ( 0 != lrc )
  {
     EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_GET_CLIENT_HANDLE,
                        0,"edmrst_get_client_handle() failure");
     p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
     return(-1);
  }

  //
  // Create an ifspec from the handle
  //
  p_psvc_ifspec = (rpc_if_handle_t *)
                  calloc(1,sizeof(rpc_if_handle_t ));
  if (p_psvc_ifspec == NULL)
  {
   EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_NO_MEMORY,
                     0,"ifspec calloc() failure");
   return(-1);
  }

  lrc = csc_private_ifspec_init( p_restore_service,
                                 EDM_DISPATCH_PROTOCOL_CLIENT,
                                 EDMDPC_FUNCTIONS,
                                 p_psvc_ifspec,
                                 &status );
  if ( 1 != lrc )
  {
    (void) free(p_psvc_ifspec);
    EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_IFSPEC_INIT_FAILURE,
                       status,"csc_private_ifspec_init() Failure");
    return(-1);
  }

  if (IsDebugOn())
  {
     EDMDispatch_logent(__FILE__,__LINE__,LOG_INFO,DDP_PORT_NUMBERS,
                        0,"PORT_INFO p_psvc_ifspec(DDCCW) port#: %d",
                        p_psvc_ifspec->portnum);
  }

  psvc_h = (rpc_binding_handle_t *) calloc(1, sizeof(rpc_binding_handle_t));
  //
  // Using the connect handle (128 bytes) received from the restore
  // service, connect to the restore service.
  //
  lrc = csc_connect_to_async_rpc_service( NULL,
                                           *p_psvc_ifspec,
                                           psvc_h,
                                           &status );
  if ( 1 != lrc )
  {
    (void) free(p_psvc_ifspec);
    (void) free(psvc_h);
    EDMDispatch_logent(
                    __FILE__,__LINE__,LOG_ERR,DDP_PRIVATE_SVC_CONNECT_FAILURE,
                       status,"csc_connect_to_async_rpc_service(
                                        ) Failure. Status is %d", status);
```

```
     return(-1);
   }

   *bh = psvc_h;
   (void) free(p_psvc_ifspec);

   return(0);
 }

/*
** ===============================================================================
**  Function:
**  Description:
**
**
**
**  Returns:     0 Sucessful
**              -1 Read Failure
** ===============================================================================
*/
int
EDMDDSvcInit()
{
struct hostent          *hp;
struct utsname          name;
error_status_t          csc_status;
  int lrc = 0;

    ELinkHandle = ELinkInitAPI(ELINK_SHELL_EDMLINK);

    if (ELinkHandle == NULL)
    {
        return -1;
    }

  //
  // Initialize the ifspec specification from the private svc
  // creation call. This call will output the DispatchDaemon_ifspec
  //
    lrc = csc_async_ifspec_init (&DispatchDaemon_ifspec,
                                 CSC_IFSPEC_PRIVATE_TYPE,
                                 DP_PROGNUM,
                                 DP_VERSNUM,
                                 (
                       dispatch_func_p_t)&edm_dispatch_protocol_service_1_table,
                                 &csc_status);


  if ( TRUE != lrc )
  {
      EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_IFSPEC_INIT_FAILURE,
                         csc_status,"csc_async_ifspec_init() Failure");
      return(-1);
  }


  //
  //  We need the system name and ip for the if_spec.
  //
  uname( &name );
  hp = gethostbyname( name.nodename );
  if ( NULL == hp )
   {
       EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_GETHOSTNAME_FAILURE,
                          0,"gethostbyname() failure");
       return -1;
    }

   ( void ) memcpy( (char*) &DispatchDaemon_ifspec.ip_addr,
                    hp->h_addr, hp->h_length );

  //
  // Register the callback functions.
  //
  lrc = csc_register_async_server_interface(
                                &DispatchDaemon_ifspec,
                                -1,
                                edm_dispatch_protocol_service_1_table,
                                edm_dispatch_protocol_service_1_nproc,
                                &csc_status );

  if ( TRUE != lrc )
  {
    EDMDispatch_logent(__FILE__,__LINE__,LOG_ERR,DDP_REGISTER_SVC_FAILURE,
                       csc_status,
                         "Failed to register asynchronous server interface.");
    return -1;
  }

    return 0;
}
```